Situational Access Control in the Internet of Things

Roei Schuster
Tel Aviv University
Cornell Tech
rs864@cornell.edu

Vitaly Shmatikov
Cornell Tech
shmat@cs.cornell.edu

Eran Tromer
Tel Aviv University
Columbia University
tromer@cs.tau.ac.il

ABSTRACT

Access control in the Internet of Things (IoT) often depends on a situation—for example, "the user is at home"—that can only be tracked using multiple devices. In contrast to the (well-studied) smartphone frameworks, enforcement of situational constraints in the IoT poses new challenges because access control is fundamentally decentralized. It takes place in multiple independent frameworks, subjects are often external to the enforcement system, and situation tracking requires cross-framework interaction and permissioning.

Existing IoT frameworks entangle access-control enforcement and situation tracking. This results in overprivileged, redundant, inconsistent, and inflexible implementations.

We design and implement a new approach to IoT access control. Our key innovation is to introduce "environmental situation oracles" (ESOs) as first-class objects in the IoT ecosystem. An ESO encapsulates the implementation of how a situation is sensed, inferred, or actuated. IoT access-control frameworks can use ESOs to enforce situational constraints, but ESOs and frameworks remain oblivious to each other's implementation details. A single ESO can be used by multiple access-control frameworks across the ecosystem. This reduces inefficiency, supports consistent enforcement of common policies, and—because ESOs encapsulate sensitive device-access rights—reduces overprivileging.

ESOs can be deployed at any layer of the IoT software stack where access control is applied. We implemented prototype ESOs for the IoT resource layer, based on the IoTivity framework, and for the IoT Web services, based on the Passport middleware.

CCS CONCEPTS

• **Security and privacy** → **Access control**; **Mobile platform security**; **Web application security**;

KEYWORDS

Access control; Internet of Things

ACM Reference Format:
Roei Schuster, Vitaly Shmatikov, and Eran Tromer. 2018. Situational Access Control in the Internet of Things. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243817

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '18, October 15–19, 2018, Toronto, ON, Canada
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243817

1 INTRODUCTION

The IoT (Internet of Things) refers to "smart" devices whose operation is governed by hardware and software controllers that communicate with each other and with Internet-based services. The IoT ecosystem includes physical devices (in particular, sensors and actuators) and software applications in industrial, medical, civic, and other settings. We focus on the consumer "smart home" devices such as thermostats, light bulbs, door locks, and smoke detectors. A quarter of U.S. homes already have at least one such device, and their prevalence is projected to grow [61].

Open-source and proprietary platforms and frameworks such as IoTivity, Nest, and SmartThings enable communication and integration among the "smart home" devices, as well as access by third-party apps. Devices such as door locks and surveillance cameras are responsible for privacy-sensitive or safety-critical functionality, thus any IoT framework must enforce device-to-device and app-to-device *access control.*

Classic access-control lists (ACLs) specify an action, a subject, an object, and the "approve or deny" decision. They are not expressive enough for the IoT, where access-control decisions often depend on the situation, context,[1] or state of the world [35, 41, 68].

1.1 Situational access control

Using situational conditions in access-control policies is a well-known approach in smartphone frameworks that predate IoT. For example, Android has a "work mode" that can be switched on by the user [63]. Starting from version 6.0, app permissions are requested (and can be revoked) dynamically to provide the user with "increased situational context" of permission use [19]. iOS apps must explicitly request a permission to access location "when the user is not interacting with the app" [62]. More generic situational access control has been often proposed for mobile OSes [9, 16, 46, 56, 72].

Many IoT frameworks track situations that are relevant to common access-control policies. For example, Nest and SmartThings track the location of the user's phone to infer whether the user is at home; Sen.se Mother, Nest, and Ecobee can also use special sensors for this purpose. SmartThings includes a "night time" indicator, updated either by scheduled routines or by the user. Amazon Echo and Google Home use "wake word" monitoring to detect when a user intends to issue a voice command.

[1] The word "context" has been overloaded and misused in the access-control literature, often appealing to Nissenbaum's theory of contextual integrity [48] to justify *any* external condition in the policies, including the state of another device (e.g., "turn on the camera if the motion detector has been activated") or even the call stack of the current function. These technical factors do not define *social* norms that govern information flow and thus have little to do with contextual integrity in Nissenbaum's sense. To avoid confusion, we use the term *situation* to refer to the environmental conditions that must be considered when making access-control decisions.

Consider GetSafe and similar mobile apps that enable users to view feeds from home security cameras on their smartphones. GetSafe can work with the Nest Cam indoor camera but requires permission from the Nest service to access it. It can also record without the user's direct involvement, e.g., when a possible burglary is detected. Since the app's primary purpose is home monitoring when the user is away, its right to access the camera should be conditioned on the "user is not at home" situation.

IoT vs. smartphone frameworks. IoT and smartphone frameworks have much in common: both control and use multiple sensors and actuators, are event-driven, have similar API access patterns, and expose sensitive data and operations to third-party apps. These superficial similarities motivate IoT access-control architectures that work the same way as in mobile OSes, with centralized reference monitors that collect all relevant situational information and make access-control decisions [11, 29, 41, 68, 71].

The critical distinction between the IoT and mobile OSes is that **access control in the IoT is fundamentally decentralized**. First, it is performed by multiple, heterogeneous frameworks with different hardware and software stacks. Second, tracking IoT-relevant situations involves interrogating (and possibly actuating) multiple devices, sensors, apps, and APIs that are not governed by the same access-control framework and require sensitive privileges. For example, inferring if the user is at home involves obtaining GPS coordinates from their smartphone and thus the ability to track the user *wherever* they go. Third, apps, the subjects of access control, are often standalone services external to the framework (e.g., GetSafe is separate from Nest but can use its API).

In a decentralized environment where multiple frameworks enforce situational access control, situation tracking should be **encapsulated** by a global interface to ensure uniform, consistent semantics and **segregated** so that its access privileges are not shared with the frameworks that use it.

Inadequacy of existing approaches. In today's IoT, situations are neither encapsulated, nor segregated. Instead, they are defined in terms of the access-control system's own information and capabilities: the embedded or available devices, user-provided configurations, and/or the execution state (open UI dialogs, method call stack, etc.) of the subject (e.g., an app) requesting access. Both research [41, 68, 72, 73] and commercial frameworks implement situation tracking as a part of the access-control system itself, coupling them so tightly that the abstract situation semantics are entangled with the low-level implementation details. For example, Nest does not have an explicit "the user is at home" condition, even though this is a common situational constraint. Instead, Nest policies interpret "the GPS coordinates of the user's smartphone and/or deployed Nest sensors indicate that the user is at a predefined 'home' location" as equivalent to "the user is at home." Overprivileged access to the user's smartphone is thus incorporated directly into the framework.

In Section 2, we explain how in addition to overprivileging, the lack of encapsulation and segregation leads to redundant and inconsistent implementations of the same functionality and forces the implementors of access control to include low-level code for communicating with other devices and interpreting their responses.

A natural solution is to separate situation tracking from the access-control policies [9, 49] by introducing dedicated situation trackers, each tasked with determining if a certain situational predicate (e.g., "the user is at home") is currently true. Prior appified environments used this idea only in centralized settings where the tracker has the same rights and capabilities as the access-control system it serves, and the interface between them is platform-specific. The former again results in overprivileging, while the latter prevents the reuse and interoperability of situation trackers across frameworks. This is a problem because, as we show in Section 2, a few common situations are responsible for the lion's share of situational constraints in real-world access-control policies.

As the semantics of situational constraints become more complex—for example, if they involve fusion of information from multiple sensors or machine learning [35]—it will be increasingly difficult to continue treating IoT access control as just another version of centralized mobile-OS access control.

1.2 Environmental Situation Oracles (ESOs)

We propose, design, and implement ESOs, situation trackers that operate at the level of the IoT *ecosystem* and are thus fully external to the access-control monitors. They expose a simple interface that access-control monitors can use to determine whether the situation is true or not. Figure 1.1 illustrates the difference between our approach and prior, centralized ones.

Like services and app frameworks, ESOs are independent units that can be directly added to and removed from the IoT ecosystem. ESOs can be explicitly incorporated into access-control policies in a manner similar to conventional permissions. A policy can specify that a subject has access to an object only if a specific situation is currently true. These constraints are visible to users and developers.

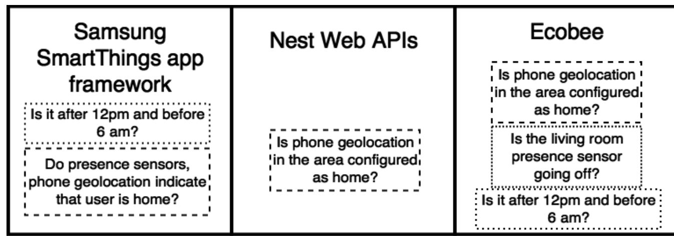
Advantages of ESOs in IoT. First, ESOs separate policy and implementation, thus enabling **two-way obliviousness** between access-control policies and situation trackers. A single ESO can serve multiple policies from different frameworks. Because policies depend only on the ESO's abstract interface and their reference monitors are oblivious of the details of the ESO's implementation, this helps ensure consistent semantics and eliminate redundancies. For example, all monitors for enforcing "allow access only if the user is at home" no longer need to query the user's phone for geolocation.

Second, ESOs encapsulate access rights needed to access other devices for the purposes of tracking a given situation (e.g., obtain geolocation from the user's phone). ESOs thus act as declassifiers and help enforce the **principle of least privilege**. Reference monitors have access only to the abstract predicate representing the situation (e.g., "the user is at home") but not to the raw data from which this situation was inferred.

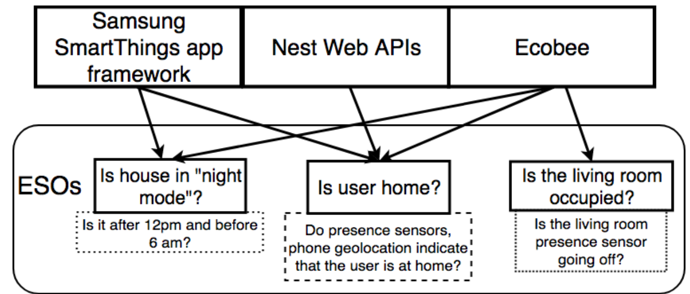
Finally, an IoT framework can support multiple ESOs for the same situation that have different semantics but expose the same API to clients. This allows developers and users to easily substitute ESOs without changing the policy and the reference monitor.

Implementation and case studies. Our ESO design makes minimal assumptions about access-control systems and is thus compatible with the various frameworks at any layer of the IoT stack.

We define the connectivity and communication protocols that ESOs must support for compatibility with the existing IoT environments. We then prototype and evaluate our approach in two different open-source frameworks representing the opposite ends



(a) Centralized approach: access control is coupled with situation tracking



(b) Our approach: ESOs separate policy and implementation

Figure 1.1: Situation tracking in access-control frameworks: centralized approach vs. our ESO-based approach

of the IoT stack: (1) IoTivity implementation of the OCF standard, and (2) Passport authorization library for Node.js. IoTivity runs on low-performance embedded devices and performs access control locally in the home network, while Node.js libraries typically run on Internet servers and mediate server-to-server communication. The main components of ESO-enabled situational access control are isomorphic in both implementations. Implementing ESOs in either layer is straightforward: hundreds of C LOCs in IoTivity, dozens of JavaScript LOCs in Passport. As case studies, we implement an ESO for tracking whether the user is at home and an ESO for logging all device accesses to a Google spreadsheet.

2 INADEQUACIES OF IOT ACCESS CONTROL

Explicit and implicit dependence on environmental situations is ubiquitous in the IoT access-control policies. For example, "the user is at home" is a very common policy constraint, supported by SmartThings, Nest, Ecobee, Wink, Apple HomeKit, Sen.se Mother, Abode, Netatmo, and Honeywell. IFTTT, a popular Web service that lets users integrate other services, including IoT frameworks, via simple "if *trigger* then *action*" recipes—confusingly called *applets* in IFTTT—provides built-in support for the "user enters an area" trigger. In the 15,000 recipes we collected from IFTTT (see Appendix A.4), "user enters an area" and "user leaves an area" are among the top three most common triggers and are typically configured to fire on home/away state changes.

As explained in Section 1, existing IoT frameworks track situations directly as part of the access-control logic. We now survey the negative effects of this policy-implementation entanglement.

2.1 Overprivileging and privacy violations

Tracking environmental situations directly as part of the access-control logic causes **overprivileging** in IoT apps and frameworks. To track whether the user is at home or away, both SmartThings and Nest rely on access to the smartphone GPS coordinates and other location sensors. This information is much more sensitive than the simple "home or away?" predicate. As a consequence, IoT apps and frameworks can persistently track the user wherever he goes, even outside the home where they are deployed. Furthermore, these **privileges are redundant**. If the user installs devices from multiple vendors in his home, all of them gain the ability to track his location anywhere in the world. The user's location is then disclosed to multiple potentially buggy or vulnerable apps, some of which may also have overly permissive data sharing policies.

2.2 Inability to enforce common policies

"Allow access but notify the user." Many users configure their IFTTT recipes to notify them about sensitive operations, e.g., when a home monitoring app initiates a camera recording.

Neither Nest, nor SmartThings supports this constraint. Their notification triggers include events such as button pushed, door opened, a family member leaves home, etc., but not the execution of a sensitive operation by an app.

"Allow access but log the operation." Even if the user is not notified in real time, some accesses need to be logged. Logging IFTTT trigger-firing events is very popular: "add row to spreadsheet" is the second most common action in our set of IFTTT recipes (see Appendix B). For example, when the SmartThings app "ridiculously automated garage door" opens the garage door, the operation should be logged. Then, if a burglary occurs, it will be possible to verify that the app did not enable this by (possibly inadvertently) letting the burglars in through the garage.

Neither Nest, nor SmartThings supports such constraints.

"Allow access only when user is not at home." The access rights of many apps—e.g., camera access by the GetSafe app from Section 1 or motion-sensor access by the smart home monitor (a popular SmartThings app)—are conditioned on the user *not* being at home.

This constraint cannot be expressed using the Nest Cam API, nor the SmartThings API. Instead, Nest tracks the "user is at home" situation and automatically turns off the camera to enforce a coarser policy: "no app can access the camera when user is at home." Apps cannot turn on the camera without the user's approval, thus *usually* they can access it only when the user is away. Critically, the user may turn the camera back on to monitor a sleeping baby or kids playing in another room or to record a family event (Nest Cam highlights these uses in its advertising). This unintentionally allows apps to resume recording even the user is at home.

SmartThings, too, supports coarse policies, with extra flexibility: situation ("mode") changes can turn any app and any device on or off. Because home monitor has to operate both when the user is at home and away, this is not useful for expressing the "allow access only when user is not at home" constraint.

"Allow access only when user is at work" Consider the IFTTT recipe "automatically post [photos] on LinkedIn when you take an instagram at work," published by LinkedIn. This recipe uses Instagram's (often inaccurate) photo geo-tagging to determine if it's Ok to post the photo. Instead, this recipe should fire only when the user is at work. The semantics would not change but would be enforced correctly. There is no way to express this in IFTTT.

"Allow access only when user is awake." The fifth and sixth most common actions in our set of IFTTT recipes are "post a tweet with an image" and "post a tweet," respectively. Consider the recipe "tweet your instagrams" that automatically tweets any Instagram photo uploaded by the user and is enabled by over 700,000 users. Tweeting is risky when the user is not in a position to respond if something goes wrong [55]. Many apps should not be able to post when the user is asleep, or watching a movie, or on a plane without Internet access, etc. There is no way to express and enforce this via Twitter and other social-media APIs.

Physical device access should also be constrained by such policies. For example, temperature-control apps should not be able to open windows if this might enable burglars to break in [41].

"Allow access only during an emergency." An app may need to change its behavior and access rights in some situations. For example, if a distress button (e.g., Flic) is pushed, an emergency app can respond by streaming live camera video to the authorities.[2] For privacy, the emergency app should not be able to access the camera when the situation "user is in distress" is not true.

"Smart" situation tracking. Different situations have different social norms. For example, when noone is in the room, potentially dangerous devices such as ovens should not be turned on automatically [74]. When kids enter a home office, it becomes unsuitable for a professional video conference [34]. When a couple is fighting, the visuals and audio are especially sensitive, requiring stricter policies. When a driver is distraught or tired, a smart car should respond [38, 67]. These situations are not easy to track, possibly requiring machine learning and aggregation of information from multiple sensors across different frameworks. Nevertheless, they can be critical factors in access control.

Situations for physical access control. In a recent study, He et al. [35] empirically show that situational factors such as user's age, device state, and proximity of other people to the device are crucial for *physical access control*, where users are subjects and device capabilities are objects. This form of situational access control requires both tracking situations and authenticating physical users. Neither is adequately supported by the existing IoT frameworks.

2.3 Inefficiencies and inconsistencies

Example: home/away. Smartphone-based geofencing is unreliable [15, 64] and inaccurate [14, 54]. SmartThings and Nest can use additional sensors, e.g., smart door locks or motion sensors, but they have different implementations and configurations and are not interoperable. Not only is the situation-tracking functionality replicated, different devices may even end up with inconsistent understanding of the environment: a SmartThings biometric door lock may recognize the user entering the home, while a Nest device, relying on inaccurate smartphone GPS, thinks the user is away.

IFTTT partially standardizes the interface for triggers, but triggers are still based not on abstract predicates such as "home or away?" but on specific implementations (the most popular one uses phone-based geofencing).

Example: user notification. The most common action in IFTTT recipes is "send a notification." User notifications can also constrain recipes: "notify me when this recipe fires" is a configuration option for *all* recipes (see Section A.4). This option is so important that it was added to the IFTTT's minimalistic configuration dialog even though IFTTT documentation exhorts recipe developers to add as few configuration fields to recipes as possible.

As in the home/away case, IFTTT-integrated services offer diverse, inconsistent implementations for the "notify me" action. In addition to the (by far the most popular) native trigger that simply sends push notification through the IFTTT phone app, there are email notifications, flicker lights, phone calls, etc. Notification methods can be very elaborate [43] and have different intended functionalities (e.g., " "notify me urgently" vs. "notify me non-obtrusively"). Recipe writers, however, have no way of selecting them by abstract functionality, only by implementation.

These redundancies raise the implementation and deployment cost of situational access-control policies such as those in Section 2.2. This cost must be paid by every framework or even every application, motivating implementers to use only coarse policies based on ad-hoc, simplistic situation trackers. Moreover, redundant, inconsistent definitions of situations hamper security and usability.

3 OUR DESIGN

We use a simple abstraction of access control, with subjects S, objects O, and a set of access rights $A \subseteq 2^O$. O are typically API-level operations, possibly subdivided by the input-parameter values if different values require different access rights.

A *policy* is a list of access-control entries (ACEs) $E \subseteq S \times A$. A reference monitor intercepts operation invocations. It allows $s \in S$ to invoke an operation $o \in O$ only if there exists $(s, a) \in E$ such that $o \in a$. Figure 3.1a depicts enforcement in this abstract framework.

We introduce **environmental situation oracles (ESOs)** as first-class objects that (1) encapsulate the tracking of environmental conditions relevant to access-control enforcement, and (2) present a uniform interface that can be incorporated into any access-control policy and invoked by any monitor in a given IoT environment.

A single ESO instantiation can be used by many client access-control systems which would otherwise all track the same situation separately. Clients no longer need to query multiple devices and interpret their responses before making access-control decisions.

ESOs are least-privilege declassifiers. They expose only essential predicates to their clients, as opposed to the underlying data (e.g., "the user is at home" vs. GPS coordinates of the user's smartphone). Access rights needed to track a specific situation are confined in a single ESO instead of propagating to all clients. ESOs are much simpler functionally than the clients that use them and thus easier to audit and update. An ESO can be maintained by a trusted party

[2]Currently, there is no popular emergency app on SmartThings. Using and developing such apps is sometimes discouraged by the SmartThings community, partly because today the app infrastructure—including the SmartApps cloud, Internet connection, and wireless connectivity—is not considered reliable enough for emergency uses.

or even by one of the client frameworks (e.g. Nest). In both cases, the other frameworks no longer need access privileges for the raw information on which the situation is based.

We argue that segregating this information inside an ESO is superior to reducing its granularity. For example, an app that receives coarse (as opposed to fine-grained) location of the user's phone is still overprivileged because it can infer information about the user well beyond the single bit revealed by the ESO.

Access-control monitors and ESOs are mutually *oblivious*, which allows for dual-faceted reusability. Multiple monitors can track the same situation using a single ESOs, and multiple ESOs (e.g., different implementations of the same environmental predicate) can be used interchangeably by the same monitor.

Writing access-control policies using ESOs is similar to writing IFTTT recipes using actions and triggers. The logic of the recipes relies on simple, uniform interfaces and is independent of the actual actions and triggers, thus a rich set of recipes can be constructed from a few triggers and actions. Similarly, access-control monitors can use the abstract interfaces of the ESOs, regardless of how the ESOs actually implement situation tracking. On the other side, IFTTT triggers (respectively, ESOs) work the same regardless of the recipes (respectively, access-control systems) that use them.

3.1 Interacting with ESOs

The interaction between an access-control system and a ESO is a basic client-server interaction.

ESO interface. Each ESO object exposes several methods. The first is *get_id*(), which returns `situation_id`, `situation_name`, and `ESO_description`. The second is *is_active*(s, a), which receives as input a subject and an access right and returns a boolean `is_active` indicating if the situation is currently active.

An ESO may optionally support a third method, *callback_on_change*(), to register callbacks for state changes. This is useful mainly for situation-triggered functionality that is not necessarily related to access control. Our survey of IFTTT recipes (Appendix B) shows that, just like situational access control, situation-triggered functionality lacks common interface standardization and suffers from overprivileging and redundancy.

Access to an ESO can be controlled, too, for example if *is_active*() and *callback_on_change*() are accessible only to authorized subjects. We use the term *e-policy* for the access-control policy of the ESO itself, as opposed to the access-control policies that are clients of the ESO. Section 3.3 explains how these e-policies are created.

Changes to the policy language. We add a set of ESO identifiers I and a set of situational ACEs (sACEs) $E_{sit} \subseteq S \times A \times I$. For every $(s, a) \in S \times A$, E_{sit} can have at most one entry pertaining to (s, a). More elaborate policies could be expressed with multiple situational constraint per subject-object pair, e.g., based on logical `and` or `or` between situations, but an ESO can aggregate query results from other ESOs, thus there is no loss of expressiveness.

These changes are backward-compatible and aim to minimize the adoption cost for the existing systems. Alternatives such as grouping rules into "profiles" that activate on situation changes or adding roles and attributes [37] are expensive to adopt in the IoT and do not appear necessary for common situational policies.

Changes to enforcement. Figure 3.1b depicts the reference monitor's behavior when enforcing a situational constraint. Upon access request $(s, o) \in S \times O$, if there exists $(s, a, i) \in E_{sit}$ such that $o \in a$, the reference monitor calls i's *is_active*() function with (s, a) as input. Access is allowed if one of the ESOs is "active." Otherwise, the reference monitor reverts to its original behavior.

If an ESO is unavailable for any reason, access is disallowed. This is a conservative choice. While it could potentially result in denial of service, the alternative is too-permissive enforcement. Either way, the user should be notified that the ESO has failed.

For some operations, access may need to be revoked and operation terminated mid-execution when the situation changes. For example, the access of a home-monitoring app to a camera may need to be terminated when the user enters the house. One solution is to use the ESO callback interface. Unavailable ESOs will not issue callbacks, however, thus client frameworks may need to periodically query the ESO and terminate the operation if the situation has changed (or the ESO has become unavailable).

Once the reference monitor has permitted an operation to start, it may not be able to terminate or even notify the operation. Operations should expose an interface to be notified by the access-control mechanism about changes in the environment—see Figure 3.1c.

3.2 Integrating ESOs into the IoT stack

In Section 4.1, we survey four IoT interfaces where access-control policies may need to be enforced: (1) direct resource-layer access, (2) service-to-service access through Web APIs, (3) app-to-service access in app frameworks, and (4) IFTTT service integrations. We categorize these interfaces by their connectivity and communication capabilities and specify the connectivity interface and communication protocol between the clients and ESOs.

Resource-layer accesses are *local*: they take place within a home network or local area network. Unlike HTTP, resource-layer protocols such as OCF, ZigBee, and Z-Wave (see Section 4) are optimized for high availability and low energy consumption. We call ESOs at this layer Resource ESOs (*rESOs*).

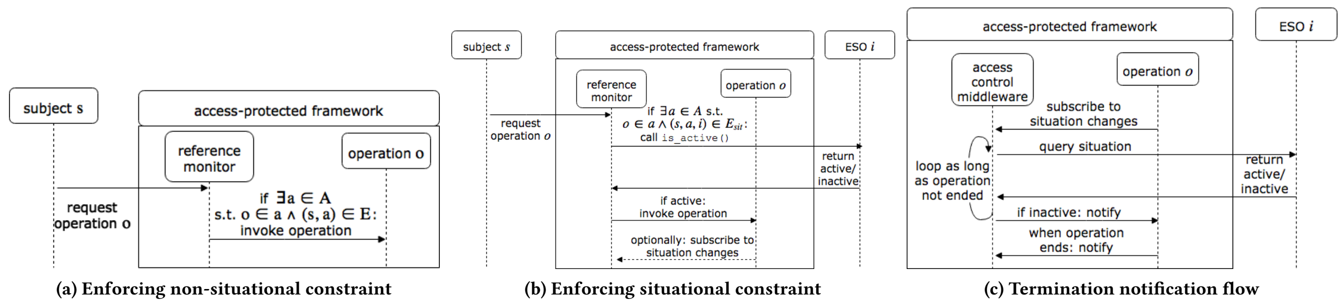
Service-to-service interactions are *Web-based*. Access-control decisions are made by a (typically remote) cloud service prior to operation dispatch. ESOs are deployed as Web services exporting an HTTP interface, i.e., Web ESOs (*wESOs*).

Monitors for app-service interactions can be Web-based (and use wESOs) or local (and use rESOs); SmartThings is a hybrid.

IFTTT recipes are deployed in the proprietary, closed-source IFTTT service. While wESOs can be used, they may not be adequate. IFTTT imposes requirements on the remote services exposing actions and triggers, e.g., they must issue OAuth 2 tokens that never expire, add integration code in IFTTT's Web IDE, and pay a fee. "Filter" code with third-party query capability (see Appendix A.4) has the benefits of an ESO for IFTTT recipes.

Our prototype implementations focus on rESOs (Section 5) and wESOs (Section 6).

Alternative: constrained delegation. For the Web, where access is delegated using tokens rather than granted based on identities, some existing delegation schemes already facilitate wESOs without any modification to the scheme's enforcement logic. In the expressive bearer-credential protocols such as SPKI/SDSI [22] and



(a) Enforcing non-situational constraint (b) Enforcing situational constraint (c) Termination notification flow

Figure 3.1: Abstract access-protected framework reference monitoring flow

(the more recent) Macaroons [6], credentials include constraints on when and how they may be used. For example, Macaroons allow adding *contextual caveats* to bearer credentials, which can include arbitrary constraints from the credential issuer or a downstream bearer, e.g., the requirement that a credential-carrying request contain a signed statement from a third party.

These access delegation schemes are expressly designed to support attestation protocols with the mutual obliviousness property of our ESOs. For example, a macaroon could carry the constraint "bearer must show evidence that they are logged into Alice's Google account" and Google could export a service that signs such statements, similar to OpenID authentication [53]. Our wESO protocol can be similarly extended to issue such statements.

3.3 Using ESOs

Creating policies with situational constraints. Different access-control systems configure policies differently. For example, in SmartThings, the user assigns access rights to individual devices during app installation (see Section A.3). Nest uses the standard OAuth 2access delegation flow. In the resource layer, policies are typically determined by a higher-level framework, not by the user.

Some frameworks already support rudimentary situational constraints, usually proprietary and with different semantics and configuration procedures, e.g., "modes" in SmartThings (see Section A.3) and "Home/Away Assist" in Nest (see Section A.2).

Our generic design does not prescribe how to generate policies with situational constraints, as this varies from framework to framework. We focus on policy enforcement rather than policy generation. ESOs can help enforce policies that are already popular but nevertheless not enforced by the existing platforms. In Section 6.3, we show how to add a wESO-tracked situational condition to any policy rule in a Web service.

Installing ESOs. rESOs can be installed into the local or home network directly by the user, but resources at this layer (and their permissions) are typically controlled by a higher-level framework that exposes a UI for setting policies. Installing a wESO is as easy as signing up for a Web service.

Installation includes configuring the ESO and and setting the e-policy for accessing the ESO's interface. Configuration is important when the ESO is based on the user-specific data and preferences (for example, home location). To specify and enforce the e-policy, rESOs can use the access-control facilities of the resource layer, while wESOs can use OAuth 2-style bearer tokens.[3]

ESOs are not exempt from their environment's access-control policies. Like any other entity in an IoT ecosystem, wESOs and rESOs may request and use access to protected APIs on the Web, within the home network, on the user's phone, etc.

Choosing ESOs. ESOs expose an interface, consisting of *get_id*() and *ESO_description*(), announcing which situation they track. This is useful for grouping and choosing from among ESOs that track the same situation, when their URIs are known.

An ESO cannot respond to these queries before it is installed and therefore cannot be discovered by potential clients. To address this challenge, we suggest that the maintainers of ESOs create *ESO providers*. Providers expose the identities and descriptions of not-yet-installed ESOs through a Web interface. The URIs of such providers can be collected and advertised by clients or other parties.

Concretely, an ESO provider exposes the *get_id*() method, which, like the corresponding method in the ESO interface, returns three values. The first two are identical to what the ESO's *get_id*() would return. The third value, `ESO_provider_description`, is an abstract description of the ESO without specific configuration values. For example, a provider that tracks the situation "it is dark outside" may produce location-specific ESOs. Its description can return "This ESO indicates if it is dark outside, inferred using date and location." For an ESO installed by this provider, the `ESO_description` can return, for example, "This ESO returns if it is dark outside in Ebbing, Missouri, United states, using the current date."

Accessing ESOs. Once an ESO has been installed, clients must know its URI. For rESOs, the resource layer broadcasts resource URIs within the home network. As long as an rESO is set as "discoverable," its authorized clients can identify and use it.[4] wESOs, however, are on the Web. Their URIs should be the same as their providers' URIs (if implemented). Either way, multiple wESO instances for multiple clients can be accessed using a single URI. Within the clients and the wESO service, different wESOs are identified and distinguished by the OAuth 2 tokens used to access them.

[3]OAuth 2 tokens are only valid for a certain duration. To avoid wESO denial of service when enforcing a situation-dependent policy, client frameworks should only grant tokens that expire before their own e-policy token for accessing the wESO service.
[4]Resources have identities coupled with cryptographic keys; resource authorization can ensure that the rESO being used is the one intended—see Appendix A.1.

In Section 6.3, we describe our prototype implementation of a Web service that lets situational constraints be added to its policy rules using wESOs and wESO providers. This includes choosing wESOs, installing them, and setting up access using OAuth 2.

3.4 Limitations

Network and performance overhead. When making a situation-dependent access-control decision, the reference monitor must query an ESO and receive an answer. This can be a problem if latency is crucial or if the access-control system or the network are request-saturated. In IoT, however, permission-protected accesses already take place over a network interface (see Section 3.2) and are thus not likely to be time-critical. Home devices and networks rarely operate near their maximal request throughput, and we expect that the capacity for inter-service HTTP requests will scale to support a large number of services and interactions.

Reliance on third parties. ESOs are third parties (vs. subjects and objects) in situational access control. An unresponsive or faulty ESO affects its clients' functionality, which is problematic if this functionality is end-to-end and cannot tolerate faults.

In practice, IoT frameworks already rely on third-party components for situation tracking (e.g., smartphone geofencing) and already deal with inconsistent and sometimes erroneous results (see Section 2.3). Therefore, ESOs are not making situation tracking worse and may even improve it.

4 THE IOT STACK

IoT devices are typically accessed via vendor-specific controller apps (*vendor apps*) that (1) configure and control devices from a given vendor and may also (2) control compliant devices from other vendors. Vendor apps may have both Web- and smartphone-based interfaces. Apps and devices from different vendors interact via high-level Web APIs and/or directly over a home network (e.g., an Amazon Echo Plus device can directly control GE lightbulbs).

We briefly survey the IoT software stack depicted in Figure 4.1, focusing primarily on the layers where access control is performed.

Layer 1: Connectivity. The connectivity layer is responsible for (1) device-to-device physical transmission and reception (often requiring a common proprietary chip), and (2) low-level device addressing and routing using IP addresses, typically using proprietary protocols that minimize energy consumption and improve the efficiency of the home network, and secure device-to-device communication facilities (e.g., DTLS sockets). Older protocols include IP over Bluetooth and Wi-Fi, newer ones include ZigBee, Z-Wave, and Thread.

Layer 2: Resources. The resource layer is responsible for (1) provisioning IoT devices, in particular assigning unique identities associated with cryptographic credentials to devices and their owners, and (2) exposing device capabilities to (possibly remote) clients and enforcing permissions when clients access the device.

The client API is relatively low-level, exposing only a few primitive methods for invoking device capabilities such as poll, update, and subscribe to the events of a device-based resource. This is akin to a distributed Linux-style file system, where files correspond to resources. Some functions of this layer are decentralized, e.g., permission checks may be performed on multiple devices, while others are implemented by centralized daemons, e.g., device provisioning.

Resource layer standardization efforts include the OCF standard, ZigBee/dotdot, Z-Wave, Apple HomeKit Accessory Protocol (HAP), Insteon, Echonet, IPSO, and others.

The resource layer does not include apps,[5] nor does it manage app installation, activation, removal, and app-identity assignment.

Layer 3: Vendor apps and services. A vendor app typically implements some functionality using the devices it can access. It can also expose a Web-based control interface through a *service*. A service runs on a cloud infrastructure and manages user accounts. Services such as Nest are operated by IoT device vendors and control devices in users' homes. Other services do not control IoT devices (e.g. Facebook, Instagram) but can interact with services that do.

Layer 4-a: Web APIs. A service may expose a Web API for querying and controlling user data and devices from the same vendor and compliant devices from other vendors. Other services can access this interface with the user's permission. This layer also defines a permission model and enforces access control. API methods and permissions are device-specific and higher-level than resource-layer primitives, e.g., "turn switch on and off." See Section A.2 for details.

Layer 4-b: (Vendor-)app frameworks. Some vendors provide frameworks that expose richer programmable interfaces and APIs to apps that are specifically designed for the framework. In addition to the APIs for device and data access, a framework can include an app programming language, an app market for publishing and distributing apps, app installation and removal interfaces, and APIs for common UI operations such as app configuration.

An example of an app framework is Samsung SmartThings, which runs apps on a dedicated hub device and provides device APIs with abstractions similar to (but less flexible and more high-level than) a smartphone app framework. Similarly, Apple's HomeKit app framework exposes an API for iOS apps to control HomeKit-compliant smarthome products. Another example is the framework for Google Assistant apps, which builds on the device's voice transcription to offer specialized functionality. For example, the Chef assistant app interactively helps users choose and follow cooking recipes. Alexa "skills" are similar.

Layer 5-a: IFTTT. IFTTT is a dedicated Web service that defines yet another abstraction for device operations and user data: *channels*, which comprise *triggers* and *actions*. Triggers and actions are provided by third-party partner services (e.g., Nest and SmartThings); a few "native" ones are provided by IFTTT itself. Triggers and actions defined by a partner service use that service's Web APIs. For example, the Instagram trigger "any new photo by you" uses Instagram's Web API to find and retrieve user's images.

Any IFTTT user or partner service can create new interactions between services by writing "if *trigger* then *action*" recipes, e.g., "publish all Instagram photos to my Facebook profile" or "shut off the smart blinds when the sun is setting."

Layer 5-b and **Layer 6** deal with third-party apps and applets.

[5]Lower-level protocol specifications can refer to the resource layer as "application layer." Some prior work [2, 28, 32, 45] incorrectly considers resource-layer implementations such as IoTivity and AllJoyn (now merged into IoTivity) as app frameworks.

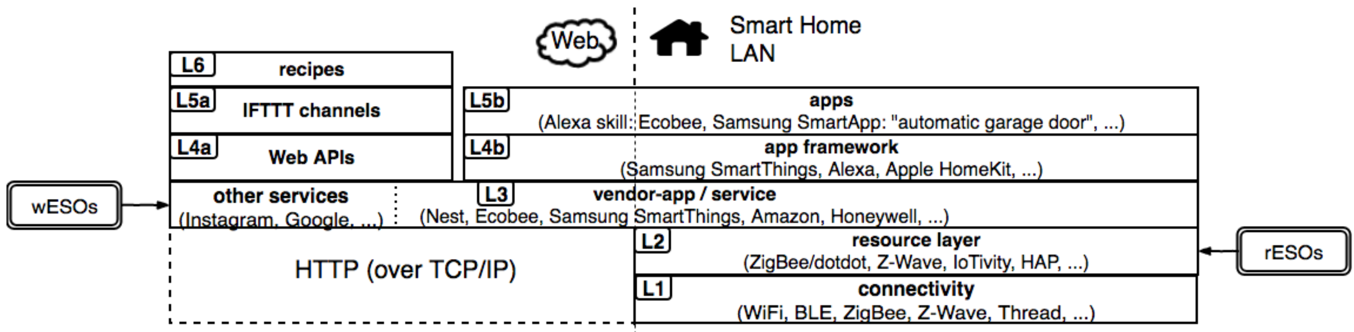


Figure 4.1: The IoT stack. wESOs export Web APIs in the service layer; rESOs are local-network resources.

Despite standardization efforts, in practice inter-vendor integration is still performed in an ad-hoc fashion. For example, Samsung SmartThings and Nest do not have an integration interface: SmartThings does not use the Nest API, and controlling Nest products via the SmartThings framework is not officially supported. This does not have a direct impact on the ESOs because they are not based on inter-vendor integration.

4.1 Access control in IoT

Four types of interactions in the IoT stack are protected by access control. We describe them generically and then provide detailed examples in Appendices A.1 through A.4.

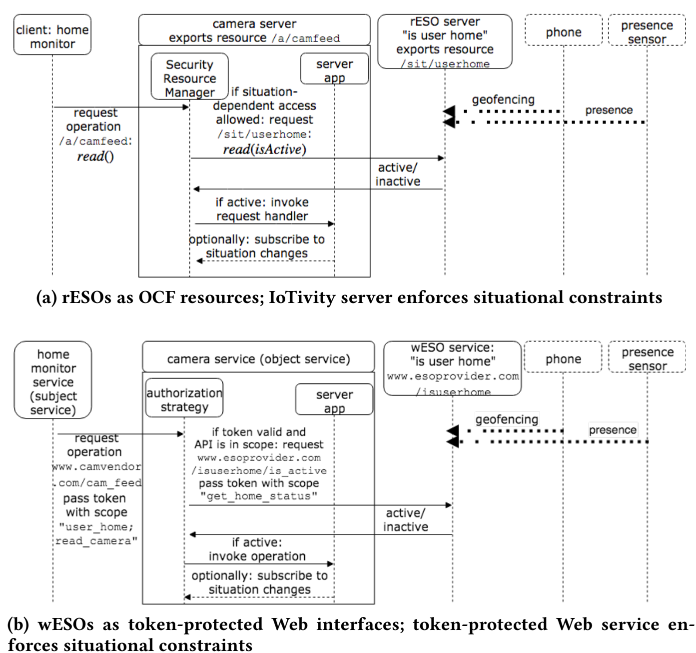
Direct resource access is typically performed by the device's "owner" app provided by its vendor. To this end, many devices implement a standardized communication protocol such as ZigBee, Z-Wave, or the OCF standard. Devices become accessible to vendor apps and other devices by acting as *resource-layer servers* that export *resources* ("endpoints" in Z-Wave, "application objects" in ZigBee). Resources can be accessed by other servers (typically, other devices) or by *resource-layer clients*, such as a hub that controls devices. Clients and servers have identities that can be used to specify resource permissions on servers. Permissions for this low-level interface are typically set by the vendor app so that they comply with its higher-level policies. Resource-layer standards do not support situational constraints at all. A representative example of a resource-layer standard is the OCF Standard (see Appendix A.1).

App-service interactions are performed via app-framework APIs such as Samsung SmartThings that may use a permission model to constrain API access. We explain SmartThings' SmartApps access control in Appendix A.3.

Service-service interactions typically involve accesses via Web APIs, although vendors can also program their respective devices to interact directly or via mobile APIs. A representative example is Nest API (see Appendix A.2).

Channel-recipe interactions are based on the service-service integrations defined in IFTTT. See Appendix A.4 for details.

As explained in Section 3.2, to serve access-control frameworks within the home network (e.g., local apps on app frameworks with direct resource-layer access), rESOs are integrated into the resource layer. To serve access-control frameworks that run on the Web, wESOs export Web APIs—see Figure 4.1.



(a) rESOs as OCF resources; IoTivity server enforces situational constraints

(b) wESOs as token-protected Web interfaces; token-protected Web service enforces situational constraints

Figure 5.1: Resource layer and Web ESOs integrated into access control in IoT.

5 RESOURCE-LAYER ESOS

rESOs can export interfaces through any resource-layer protocol such as ZigBee, Z-Wave, or the OCF standard. The three standards have similar abstractions. For Z-Wave, we were unable to locate public details about access-control standardization (if any). In ZigBee, access control is supposedly standardized by a recently advertised specification, "dotdot", which is not publicly available. Therefore, we focus on the OCF standard and its open-source implementation, IoTivity. IoTivity is supported and maintained by hundreds of IoT vendors, with major companies such as Microsoft, Samsung, Intel, Qualcomm, and others leading the implementation.

Our rESOs are resources registered by a server process. Their query interfaces, with the *get_id*() and *is_active*() methods, are exported via the permission-protected `read` operation and can be accessed only from authorized clients (see Appendix A.1). rESOs

can be installed and configured directly by users, but servers are typically installed by a higher layer, i.e., a service or app framework.

5.1 Situational access control in IoTivity

IoTivity, coded in C and C++, facilitates OCF client and server development. For clients, it supports APIs for issuing asynchronous requests; for servers, it supports an API for resource registration, allowing the server to associate resource identifiers with request handlers. IoTivity handles associating requests with registered resource methods, as well as access control. Every client and server must provide a JSON file with configuration resource values.

We changed IoTivity's implementation of an OCF server to support ESO-based situational access control as described below.

Server enforcement before change. Incoming requests from the network are handled asynchronously. First, the communication middleware pushes them onto a task queue. One or more handler threads sequentially pulls requests from the queue and calls into the secure resource manager (SRM), which is the IoTivity reference monitor. The SRM constructs a *request context* object to track requests throughout the process of authorization and execution. This object contains the authenticated identifier of the remote requester and network-level identifiers, whether or not the request was granted, and other metadata (e.g., the accessed method and URI). It then invokes the *CheckPermission*(*requestContext*) method, which has the following logic. First, it checks if the request is from the device owner or resource owner for configuration resources, and if so, approves. Then, it iterates over ACEs and tries to find a match for the object and subject. If the request is approved, the SRM calls into the resource access handler.

Server enforcement after change. We changed the JSON schema for `/acl` configuration, as well the corresponding parsing code, to add an optional field to every ACE that specifies the situation identifier. If the ACE contains a situation identifier, the request context is passed onto the *situation client* (SC) module that handles interaction with the rESO. SC constructs an *is_active*() request to the rESO, specifying the subject identity (GUID) and access right request (URI, access type, and device identity).

For this solution to scale to multiple concurrent requests, we implemented two crucial optimizations. First, because response time is contingent on the rESO, queried an acknowledgement message is sent to the requesting client before the rESO is queried. It signals that the request is processing and retransmissions are unnecessary. Second, the situation query is asynchronous: right after the remote request is issued, we yield the thread so that concurrent requests can be handled. When a response is received or after a (configurable) timeout, if the situation is not confirmed as active—the rESO responded that it is inactive, access to the rESO was denied, or the request timed out—*CheckPermission*() continues to try to find a matching valid ACE. Otherwise, it finishes, and the SRM flow continues to perform the request. Figure 5.1a depicts this flow.

Operation termination. After a situation-dependent approval has been issued for a request, SC maintains a handle to a `termination` object and passes it through the SRM to the request handler. Using this shared object, the request handler can indicate to the SC that periodic querying is necessary and provide a termination callback. It can subsequently signal to the SC that the operation has ended organically and there is no need to query further.

5.2 Micro-benchmarks

Code changes. Our implementation of the SC module totals 824 LOCs of C. Main changes in IoTivity itself total 442 line insertions and 147 line removals.

The main code changes required in IoTivity were (1) passing the termination handle from the SC to the request handler, (2) adding the rESO address to the policy's ACE JSON schema, and (3) splitting the SRM logic into asynchronous handlers to facilitate sending a request to the rESO during a permission check without blocking the handler thread. The last change, which is also the most challenging (and requires most LOCs) is already planned for IoTivity's SRM, to support querying a centralized security manager for some requests. Because it has not yet been integrated into IoTivity's mainline branch, we added it ourselves.

Experimental setup. To measure the added overhead in permission checks when querying rESOs, we used an IoTivity client on an i7-5960X machine running Ubuntu as our subject. The object is an OCF server exposing the `/a/cam` resource that responds to a `read` method, simulating an IoT device interface. We installed our server on a Raspberry Pi Model B with an ARMv7 Processor rev 4 running Raspbian (this low-performance setup is similar to an IoT embedded device). Our prototype rESO contains `/a/is_user_home`. The rESO server was installed on an Ubuntu Intel E5-1660 machine. The machines were all connected to the same LAN.

Our subject accesses `/a/cam`'s `read` operation in an infinite loop. In the following experiments, we compare the timings in two settings: (1) the "vanilla" setting, where the object server configuration contains an ACE that grants the subject read permission, and (2) the situation-query setting, where the read permission is granted conditional on the rESO's approval. The only difference between the settings is that in (2), the ACE specifying client permissions to `/a/cam` also specifies a situational constraint in the `cnd` field:

```
"aclist2": [ ... {
  "aceid": 3,
  "subject": { "uuid": "CLIENT_GUID" },
  "resources": [{ "href": "/a/cam",
    "cnd":"ESO_GUID:/a/is_user_home" }],
  "permission": 7 } ]
```

We launched multiple concurrent instances of our subject and, over the course of 3 minutes, measured the duration of the method *CheckPermission*() for `/a/cam`, average round-trip time for a request to `/a/cam`, and the average processor load of `/a/cam`'s IoTivity process as the percentage of a single core's capacity sampled at 2-second frequency via `top`. Table 1 summarizes the results.

Results. On average, 301 requests per minute were sent and handled when permission checks involve situational queries, compared to 528 for the vanilla setting. Permission checks that involve querying an rESO consistently take similar average time, with up to 15% variance, as querying the vanilla `/a/cam`. The average round-trip time for requests to `/a/cam` is consistently almost twice the average time for checking permissions. This is expected, because the

#clients	V/RTT	V/proc	V/CP	SQ/RTT	SQ/proc	SQ/CP
1	0.162	11.47%	0.0005	0.254	17.613%	0.128
5	0.512	21.16%	0.0006	0.937	23.543%	0.473
10	1.033	20.915%	0.0009	1.856	25.348%	0.938
20	2.075	21.123%	0.0009	3.676	24.351%	1.841

Table 1: rESO micro-benchmark results for concurrent access requests issued for 3 minutes in the situation query (SQ) and vanilla (V) settings. Round trip (RTT) and $CheckPermission()$ **times (CP) are in seconds.**

#clients	V/RTT	V/proc	V/CP	SQ/RTT	SQ/proc	SQ/CP
1	0.0185	92.1%	0.006	0.038	67.3%	0.025
5	0.0635	100.9%	0.016	0.093	105.7%	0.056
10	0.1228	99%	0.034	0.182	102.2%	0.103
20	0.2052	100.1%	0.06	0.326	105.5%	0.191
40	0.3960	101.9%	0.114	0.593	109.6%	0.365
80	0.7704	102.1%	0.222	1.178	108.1%	0.737

Table 2: wESO micro-benchmark results. The situation query (SQ) and the vanilla (V) settings; concurrent access requests issued for over 3 minutes. Round trip (RTT) and $CheckPermission()$ **(CP) times are in seconds.**

overhead of permission checks using an ESO is dominated by the additional `/a/is_user_home` request RTT (see Section 3.4).

We further observed that the processor load increased by about 5% for the situation-query setting. This can be explained by the additional request and acknowledgement messages, as well as the cost of making permission checks asynchronous, i.e., saving and reinstating the request-handling state before and after the situation query. This overhead can be reduced by optimizing the SC.

Note that, while this benchmark is useful for verifying implementation sanity and grounding assessments of overheads in many scenarios, it does not directly measure overheads in all ESOs; those may vary with network conditions and ESO processing time.

6 WEB ESOS

Every Web ESO (*wESO*) is provided by a Web service, which is independent of the object and subject services. The wESO's interface, including the two methods exported by all ESOs, is accessed via HTTP GET calls to its URI with the appropriate method name and accompanied by an appropriate bearer token. The results are returned in JSON-encoded responses. See Appendix A.2 for more details and an example involving Nest Cam APIs.

6.1 Situational access control in a Web service

To support ESO-based situational access control on the Web, we changed an **object service**, which is implemented in JavaScript and runs over Node.js servers with Mongoose DB schemas. Following common practice, the object service uses bearer tokens, verified using the popular Passport authentication library, to control access to its single API, $get_user_info()$.

Passport enforces access control as follows. When protected APIs are accessed, an *authorization strategy* is called. This function is defined once but can be used for multiple APIs. It receives the request, which includes the caller's credentials, the access rights required for the protected API, and a callback implementing the API's behavior. The strategy first decides whether to deny or allow access. Strategies usually follow common patterns, such as username-and-password-based "local" authorization or bearer tokens. The passport-http-bearer library offers middleware for the latter. It verifies that the token is valid (i.e., was issued by the object service), non-expired, and that its scope allows the invoked API call. The strategy then finds the user profile associated with the request, and, if access is allowed, calls the function that implements the API's behavior, passing the user profile to it.

Server enforcement after change. Internally, ACEs are expressed using access tokens that are linked in the service database to a wESO's URI and access token. The *scope* of the token specifies access rights encoded as a list of *permission strings*—see Appendix A.2 for more details. We add the situation ID as a prefix to every permission string, followed by a delimiter (we used ";"). Changing the scope to reflect situation dependence is important since, in OAuth 2, the scope is sent to subjects to specify their access rights.

We implemented our strategy middleware, passport-http-bearer-ESO, as a fork of the passport-http-bearer library. When used in lieu of the original passport-http-bearer, it performs authentication as follows. When a token linked to an ESO is received in a request, it is first verified using the vanilla authorization logic, which also extracts the user's profile. The associated ESO is then queried using a GET request to the ESO's URI (with the received token). If the ESO is active, the requested API is invoked and passed the user's profile, as well as a situation-change subscription function (see below). Figure 5.1b depicts the enforcement logic.

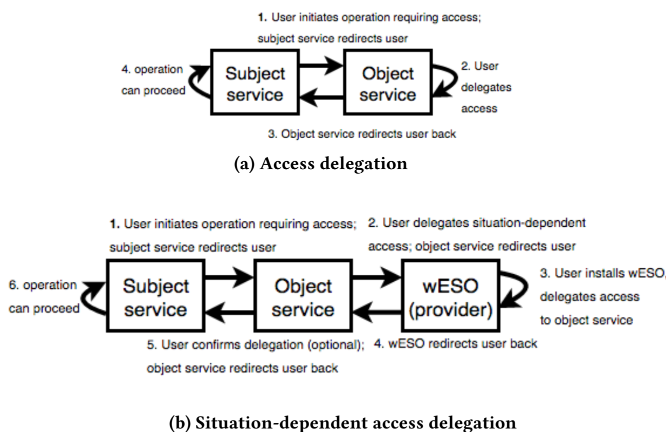
Operation termination. The situation-change subscription function receives as input a termination callback (to be called if the situation is no longer active, or the ESO is irresponsive), and the time to wait between periodic situation queries. It initiates periodic queries and calls the termination callback if the ESO is no longer active. The subscription function returns a callback for the API to invoke when the operation stops (and so should periodic querying).

6.2 Micro-benchmarks

Code changes. The entire enforcement behavior is contained in the authorization strategy implemented by our passport-http-bearer-ESO library, which adds 56 LOC to passport-http-bearer. Services that use the latter can simply use our library instead. This involves changes to **3 lines of code**, to change the strategy name and add ESO URIs and access tokens to the persistent token registry.

Experimental setup. We measure request times similarly to Section 5.2: a *subject service* initiates periodic concurrent queries to saturate the handling capacity of the object service and measures times for permission checks, processor load, and request round-trip times. The subject, object, and ESO are all Node.js server processes running on an Intel i7-5960 PC, a Raspberry Pie Model B, and an Intel E5-1660 respectively. The queried object-service API is `/info`, which, after token authorization, simply returns the processor load and time that authorization took.

Results. The results are similar to Section 5.2, except the (more mature) Node.js infrastructure demonstrates overall much better performance than IoTivity. For 5-40 concurrent requests, on average



(a) Access delegation

(b) Situation-dependent access delegation

Figure 6.1: Situational vs. non-situational access delegation. Users instantiating operations that require access rights are referred to another website, where they configure and delegate access, and then back to the original website to continue with the original flow. For situational access delegation, an additional, nested wESO instantiation flow may occur.

3,300 requests per minute were sent and handled by the situation-querying server, compared to about 5,200 for the vanilla server. Both seamlessly handle up to 80 concurrent requests.

6.3 Creating situational policies on the Web

In Section 6.1, we added a situational constraint to an OAuth 2-protected API. To enforce this constraint, the object service used a hard-coded wESO and its access token. We now explain how the user can select an arbitrary wESO to situationally constrain access to the object service by subject services.

Delegating access using OAuth 2. Our object service uses OAuth 2 to enable the user to delegate access to its API. At a high level, when subject services access the protected API, they redirect the user (using an HTTP redirect response) to the OAuth 2 entry point at the object service. The user can then approve access and is redirected back. See Appendix D for more details.

Adding situational constraints. We changed our object service so that, when the user is approving access, she can (1) constrain it using a wESO of her choice, and (2) provide the token for this wESO to the object service. To this end, a second, nested OAuth 2 flow redirects the user to a wESO server, which exposes the wESO-provider interface (see Section 3.3) and protects wESO APIs using OAuth 2 “authorization code grant.” User can choose from the existing wESOs or authorize the installation of a new one.

Figure 6.1 depicts standard access delegation in the original object service and situation-dependent access delegation in our modified object service with a nested wESO instantiation flow.

Choosing a wESO. In step (2) of Figure 6.1b, the user chooses a wESO service URI (see an illustration in Figure 6.2). The object service displays the description of the wESO provider. If the object service already has tokens to access wESOs with this URI, it displays the corresponding wESO descriptions. If the user’s desired wESO is on this list, she can choose it, obviating the nested flow.

Connect with *home-monitor-app.com*

Signed in as **Alice** (not Alice?)

Click *Authorize* to allow ***home-monitor-app.com*** to **use indoor camera**

Allow situation-conditioned access, using a situation tracker (wESO).

wESO provider: ***www.esoprovider.com/isuserhome*** (change wESO provider?)

Set up a new situation: Home presence detection using phone geolocation

Or use an already configured situation with this provider:
- Home presence detection for The House at Roosevelt Island, New York City, using phone geolocation
- Home presence detection for Apartment at Manhattan, New York City, using phone geolocation

Authorize →

Figure 6.2: Choosing a wESO: step (2) of Figure 6.1b.

Implementation. Implementing the above behavior is easy in any Web service that protects APIs with OAuth 2. Nested flows are the biggest challenge because they involve two concurrent “authorization code grant” OAuth 2 flows: the encapsulating one, where the object service issues a token, and the nested one, where the wESO service issues a different token to the object service. Concretely, at step (2) of Figure 6.1b, we save the encapsulating flow’s state using a Mongoose DB schema; at step (4), we reload it and present a confirmation dialog to the user; and at step (5), we finish the nested OAuth 2 flow, continue the encapsulating flow, and eventually redirect the user to the subject service. The last step requires exchanging an “access code” for a token with the wESO service and issuing an access token for the subject service (see Appendix D). These changes total 122 LOC in JavaScript. The confirmation screen was implemented with a 45-line Jade scheme.

Like the authorization strategies in Section 6.1, all changes are local. The OAuth 2 flow that we modify is typically implemented once per service—but can be invoked with different scopes, to delegate access to different sets of APIs and operations.

6.4 Case studies

We describe Web ESOs that track situations motivated by the examples in Section 2: “allow access but log the operation,” “allow access only when the user is not at home,” and “allow access only when the user is in the room.” For each situation and ESO, we specify: (1) the return values of *get_id*, (2) the protected APIs used by the ESO, (3) ESO configuration values, (4) its offline behavior and the behavior of *is_active*. We implemented the first two ESOs, which track common situations. The third ESO tracks a situation not commonly available in IoT frameworks but still has interesting use cases.

Appendix C also describes an alternative ESO for detecting the “user is away” situation using Nest and SmartThings presence sensors, and two ESOs for notifying the user via (a) push notifications, and (b) push notifications and blinking SmartThings lights.

Log accesses to a Google Drive spreadsheet. **Identity values:** `{situation_name: logAction; ESO_description: "log access operations to a Google Drive spreadsheet."}`. **Protected APIs, configuration:** This ESO uses Google Drive with the scope `spreadsheets`. It is configured with the name of the spreadsheet and sheet to use. **Behavior:** *is_active*(s, a) logs s’s access to a with

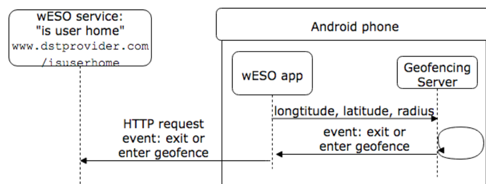


Figure 6.3: Geofencing-based "home/away" detection

a timestamp and returns "active" only when the log operation succeeds (it is "fail-safe"). **Use cases:** In Section 2.2, we explain why this is a particularly useful ESO. **Delay induced by logging:** We used the setting from Section 6.2 but extended with this ESO to measure the delay in access operations due to the ESO constraint. When performed from our campus network, the logging operation takes about **0.5 seconds** (averaged over 60 requests).

User is away, using Android geofencing. **Identity values:** `{situation_name: HomeAway; ESO_description: "Home presence detection using phone geolocation."}`. **Protected APIs, configuration:** User is required to install an Android app that uses location services. The app requires the `ACCESS_FINE_LOCATION` and `WAKE_LOCK` permissions, the latter to keep the CPU from entering sleep mode when handling location updates. The user configures the location of their home in the app. **Behavior:** the Android app uses the geofencing API [18] to track if the user is in close proximity to their home (a location configured by the user). To this end, it registers a listener ("pending intent") with the geofencing server on the device, for the event of the user coming within 600 feet of their home coordinates. On server updates, the ESO backend is updated via HTTP—see Figure 6.3. $is_active(s, a)$ returns `true` when the app indicates that the user is away.

Family member in the room, using OpenCV face detection and recognition. `{situation_name: PersonDetector; ESO_description: "Apply a face classifier to captured video to detect when specific family members are in the room."}` **Protected APIs, configuration:** requires an OAuth 2 access token to the Nest Camera, with access scope `camera_read`. Configuring the ESO include supplying several images of family members to train the classifier. **Behavior:** first identify a face in the camera feed using OpenCV's face detection API, then use the classifier trained to identify faces of family members. $is_active(s, a)$ returns "active" when a family member is present. **Use cases:** For example, a home monitoring camera can be used to record nannies, cleaning personnel, delivery personnel [59], and any untrusted person around the house, but only when a trusted family member is not in the room. It could also detect children in the room and revoke access from a videoconferencing app (see Section 2.2).

7 RELATED WORK

Access control in appified environments. Many proposals for tightening security policies in appified environments make assumptions about the apps' structure or runtime. Some build on language-specific static analyses [3, 4, 24, 27, 33, 42] or dynamic analyses in Android's Dalvik/Art virtual machine [23, 33, 52] to detect and prevent unwanted information flows. Static analysis has been suggested for SmartThings, too [11]. Other works employ Android's inter-process communication and/or kernel access monitors to mitigate collusion and confused deputy attacks [7, 8, 10, 21, 26] for mandatory access control [10, 60] and information flow control (IFC) [36, 40, 69]. For SmartThings, runtime monitoring-based IFC [29] and data provenance collection [71] have been considered. Other approaches monitor the kernel [10, 60].

NLP-based approaches have been proposed to infer the desired policy from the app-market descriptions for Android [50] and SmartThings [68] apps.

Wijesekera et al. [72] found that subdividing dynamically granted permissions according to the runtime information (e.g., "is the app in the background?") can sometimes be effective in balancing fine granularity of permissions and the need to prompt the user. PmP [13] follows this approach but focuses on informing the users about access requests from third-party libraries. Wijesekera et al. [73] use even finer-grained permissions but offload some of the decisions to machine learning classifiers. In the IoT domain, ContexIoT [41] subdivides dynamically granted permissions for SmartThings SmartApps according to the app's control flow and the source of the data used in permission-protected operations.

In IoT, however, apps (subjects) are typically entirely external to the access-control system, which thus cannot monitor their execution. Their code is often proprietary, and they are not built in a specific programming framework or distributed through a centralized app market that can facilitate inspection and vetting.

Situational access control. Dating back to the early 2000s, multiple papers suggested designs for role-based access control (RBAC) systems that add explicit situation-dependent constraints to policies. These are specific to the RBAC setting [17, 49, 51] and most require situation tracking to run locally [5, 44, 51, 75, 76].

Explicit situation-dependent constraints for Android have been proposed in CRePE [16], MOSES [56], FlaskDroid [9], Shebaro et al. [57], and Apex [46]. Policy rules are activated and de-activated dynamically by configurable detectors ("context detectors" in CRePE and MOSES, "dynamic constraints" in Apex, "context providers" in FlaskDroid and Shebaro et al.). These approaches all assume that situation tracking is done on the Android platform and using its capabilities. In particular, they define situational predicates using Android sensors. In CRePE, authorized third parties activate situations via SMS. In FlaskDroid, trackers are special plugins running on the Android framework API. None of these approaches support the encapsulation and segregation required for the IoT situation trackers (see Section 1.1).

For IoT, Yu et al. [74] argue for monitoring access requests and enforcing situational policies at the network level rather than on the devices. This is complementary to our ESO-based approach.

Other attacks and defenses in IoT frameworks. Many works focus on IoT security issues other than access control. Fernandes et al. [28] found flaws in the SmartThings app security model that lead to overprivilege and demonstrated the resulting attacks. Apthorpe et al. [1] analyze devices' encrypted traffic and suggest defenses. Simpson et al. [58] propose a hub-based system for detecting and preventing vulnerability exploitation in IoT devices. FACT [45] detects "functionalities" that devices and resources support, isolates

them from one another, and enforces functionality-level rules. Surbatovich et al. [66] show how IFTTT configuration can introduce non-intuitive, unexpected information flows, and that users often define and use recipes that not only enable but also automate and streamline privacy and integrity violations. DTAP [30] offers provenance verification for triggers in trigger-action platforms (such as IFTTT) while substantially reducing their required privileges.

Delegated access control. Constrained bearer-credential protocols such as SPKI/SDSI [22] and Macaroons [6] are complementary to ESOs. They can be used as alternative enforcement modes that query ESOs in delegation-based systems, as discussed in Section 3.2.

8 CONCLUSION

We identified a fundamental problem with situational access control in today's IoT: situation tracking is entangled with the enforcement of access-control policies. This leads to overprivileging, inefficiency, and inability to enforce common policies. The root cause is the design of the existing IoT access-control frameworks, which view IoT as a centralized platform similar to a smartphone app framework.

We proposed and implemented environmental situation oracles (ESOs) as a simple and generic solution suitable for access control at all layers of the IoT software stack, and concretely demonstrated the benefits of ESOs with prototypes for the resource layer and Web-services layer of the IoT stack.

Acknowledgements. Roei Schuster and Eran Tromer are members of the Check Point Institute for Information Security. This work was also supported by the Blavatnik Interdisciplinary Cyber Research Center (ICRC); DARPA and ARO under Contract #W911NF-15-C-0236; Israeli Ministry of Science and Technology; Leona M. & Harry B. Helmsley Charitable Trust; Schmidt Sciences; and NSF awards 1423306, 1445424, 1611770, and 1612872. Thanks to Tom Tytunovich for sharing his expertise in Web backend technologies.

REFERENCES

[1] Noah Apthorpe, Dillon Reisman, Srikanth Sundaresan, Arvind Narayanan, and Nick Feamster. Spying on the Smart Home: Privacy Attacks and Defenses on Encrypted IoT Traffic. *arXiv preprint arXiv:1708.05044 2017*.

[2] Stefan-Ciprian Arseni, Simona Halunga, Octavian Fratu, Alexandru Vulpe, and George Suciu. Analysis of the Security Solutions Implemented in Current Internet of Things Platforms. In *IEEE Grid, Cloud & High Performance Computing in Science (ROLCG) 2015*.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269.

[4] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using Static Analysis for Automatic Assessment and Mitigation of Unwanted and Malicious Activities within Android Applications. In *Malicious and Unwanted Software (MALWARE) 2011*.

[5] Rafae Bhatti, Elisa Bertino, and Arif Ghafoor. 2005. A Trust-based Context-aware Access Control Model for Web-services. *Distributed and Parallel Databases* 18, 1 (2005), 83–105.

[6] Arnar Birgisson, Joe Gibbs Politz, Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Network and Distributed System Security Symposium (NDSS) 2014*.

[7] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. XmAndroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04 2011*.

[8] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Network and Distributed System Security Symposium (NDSS) 2012*.

[9] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *USENIX Security Symposium 2013*.

[10] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Towards a Framework for Android Security Modules: Extending SE Android Type Enforcement to Android Middleware. *Technische Universität Darmstadt, Technical Report TUD-CS-2012-0231 2012*.

[11] Z Berkay Celik, Leonardo Babun, Amit K Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive Information Tracking in Commodity IoT. *arXiv preprint arXiv:1802.08307 2018*.

[12] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth Demystified for Mobile Application Developers. In *ACM Conference on Computer and Communications Security (CCS) 2014*.

[13] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I Hong, and Yuvraj Agarwal. Does this App Really Need My Location?: Context-Aware Privacy Management for Smartphones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT) 2017*.

[14] SmartThings Community. 2016. Fix for Android Presence Arrival Problems. https://community.smartthings.com/t/fix-for-android-presence-arrival-problems/61021. (2016). Accessed: Jan 2018.

[15] SmartThings Community. 2018. The Many Ways of Detecting Presence. https://community.smartthings.com/t/faq-the-many-ways-of-detecting-presence/51563. (2018). Accessed: Jan 2018.

[16] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: Context-Related Policy Enforcement for Android. In *Information Security Conference (ISC) 2010*.

[17] Michael J Covington, Wende Long, Srividhya Srinivasan, Anind K Dev, Mustaque Ahamad, and Gregory D Abowd. Securing Context-aware Applications Using Environment Roles. In *ACM Symposium on Access Control Models and Technologies (SACMAT) 2001*.

[18] Android Developer. 2018. Android Geofencing API. https://developer.android.com/training/location/geofencing.html. (2018). Accessed: March 2018.

[19] Android Developer. 2018. Permissions in Android 6.0. https://developer.android.com/training/permissions/usage-notes.html. (2018). Accessed: Feb 2018.

[20] Facebook Developers. 2018. Facebook Permissions. https://developers.facebook.com/docs/facebook-login/permissions/. (2018). Accessed: 2018-01-08.

[21] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX Security Symposium 2011*.

[22] Carl M Ellison. 2011. SPKI. In *Encyclopedia of Cryptography and Security*. Springer, 1243–1245.

[23] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems (TOCS) 2014*.

[24] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium 2011*.

[25] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *ACM Symposium on Usable Privacy and Security (SOUPS) 2012*.

[26] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium 2011*.

[27] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *ACM International Symposium on Foundations of Software Engineering (SIGSOFT) 2014*.

[28] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security Analysis of Emerging Smart Home Applications. In *IEEE Symposium on Security and Privacy 2016*.

[29] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security Symposium 2016*.

[30] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *Network and Distributed System Security Symposium (NDSS) 2018*.

[31] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decoupled-IFTTT: Constraining Privilege in Trigger-Action Platforms for the Internet of Things. *arXiv preprint arXiv:1707.00405 2017*.

[32] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Security Implications of Permission Models in Smart-Home Application Frameworks. *IEEE Symposium on Security and Privacy 2017*.

[33] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Network and Distributed System Security Symposium (NDSS) 2015*.

[34] The Guardian. 2017. BBC Interviewee Interrupted by His Children on Air. https://www.theguardian.com/media/video/2017/mar/10/bbc-correspondent-interrupted-by-his-children-live-on-air-video. (2017). Accessed: Jan 2018.

[35] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. Rethinking Access Control and Authentication for the Home Internet of Things (IoT). In *USENIX Security Symposium 2018*.

[36] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren't the Droids You're Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *ACM Conference on Computer and Communications Security (CCS) 2011*.

[37] Vincent C Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. 2013. Guide to Attribute Based Access Control (ABAC) Definition and Considerations (Draft). *NIST Special Publication* 800, 162 (2013).

[38] Xiping Hu, Junqi Deng, Jidi Zhao, Wenyan Hu, Edith C-H Ngai, Renfei Wang, Johnny Shen, Min Liang, Xitong Li, Victor Leung, and Yu-Kwong Kwok. SAfeDJ: A Crowd-Cloud Codesign Approach to Situation-Aware Music Delivery for Drivers. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM) 2015*.

[39] IETF. 2018. The OAuth 2.0 Authorization Framework. https://tools.ietf.org/html/rfc6749. (2018). Accessed: Jan 2018.

[40] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time Enforcement of Information-flow Properties on Android. In *European Symposium on Research in Computer Security (ESORICS) 2013*.

[41] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z Morley Mao, and Atul Prakash. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *Network and Distributed System Security Symposium (NDSS) 2017*.

[42] Jinyung Kim, Yongho Yoon, and Kwangkeun Yi. ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. *Mobile Security Technologies (MoST) 2012*.

[43] Thomas Kubitza, Alexandra Voit, Dominik Weber, and Albrecht Schmidt. An IoT Infrastructure for Ubiquitous Notifications in Intelligent Living Environments. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp): Adjunct 2016*.

[44] Devdatta Kulkarni and Anand Tripathi. Context-Aware Role-Based Access Control in Pervasive Computing Systems. In *ACM Symposium on Access Control Models and Technologies (SACMAT) 2008*.

[45] Sanghak Lee, Jiwon Choi, Jihun Kim, Beumjin Cho, Sangho Lee, Hanjun Kim, and Jong Kim. FACT: Functionality-centric Access Control System for IoT Programming Frameworks. In *ACM Symposium on Access Control Models and Technologies (SACMAT) 2017*.

[46] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *ACM Conference on Computer and Communications Security (CCS) 2010*.

[47] Nest. 2018. Works with Nest. https://workswith.nest.com/. (2018). Accessed: Jan 2018.

[48] Helen Nissenbaum. 2009. *Privacy in Context: Technology, Policy, and the Integrity of Social Life.* Stanford University Press.

[49] OASIS. 2018. XACML Specification. http://docs.oasis-open.org/xacml/. (2018). Accessed: March 2018.

[50] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *USENIX Security Symposium 2013*.

[51] Mor Peleg, Dizza Beimel, Dov Dori, and Yaron Denekamp. 2008. Situation-based Access Control: Privacy Management via Modeling of Patient Data Access Scenarios. *Journal of Biomedical Informatics* 41, 6 (2008), 1028–1040.

[52] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *ACM Conference on Data and Application Security and Privacy (CODASPY) 2013*.

[53] David Recordon and Drummond Reed. OpenID 2.0: A Platform for User-centric Identity Management. In *ACM Workshop on Digital Identity Management (DIM) 2016*.

[54] Reddit. 2017. Galaxy S8 reddit: Problems with IFTTT Location Triggers. https://www.reddit.com/r/GalaxyS8/comments/69tw7q/problems_with_ifttt_location_triggers/. (2017). Accessed: Jan 2018.

[55] Jon Ronson. 2018. How One Stupid Tweet Blew Up Justine Sacco's Life. https://www.nytimes.com/2015/02/15/magazine/how-one-stupid-tweet-ruined-justine-saccos-life.html. (2018). Accessed: March 2018.

[56] Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. MOSES: Supporting Operation Modes on Smartphones. In *ACM Symposium on Access Control Models and Technologies (SACMAT) 2012*.

[57] Bilal Shebaro, Oyindamola Oluwatimi, and Elisa Bertino. Context-based Access Control Systems for Mobile Devices. *IEEE Transactions on Dependable and Secure Computing (TDSC) 2015*.

[58] Anna Kornfeld Simpson, Franziska Roesner, and Tadayoshi Kohno. Securing Vulnerable Home IoT Devices with an In-hub Security Manager. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom) Workshops 2017*.

[59] Slashdot. 2017. Amazon Keys Puts Deliveries — and Delivery People — In Your Home. https://news.slashdot.org/story/17/10/25/1813258/amazon-key-puts-deliveries----and-delivery-people----in-your-home. (2017). Accessed: 2017-07-18.

[60] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Network and Distributed System Security Symposium (NDSS) 2013*.

[61] Statista. 2018. Smart Home Market Penetration in US. https://www.statista.com/outlook/279/109/smart-home/united-states. (2018). Accessed: 2018-08.

[62] Apple Support. 2017. iOS: Allow Location Access Only While Using App. https://support.apple.com/en-gb/HT203033. (2017). Accessed: 2017-07-18.

[63] Google Support. 2017. Android Work Mode. https://support.google.com/work/android/answer/7029561?hl=en. (2017). Accessed: 2017-07-18.

[64] SmartThings Support. 2018. Known Mobile Presence Issues. https://support.smartthings.com/hc/en-us/articles/204744424. (2018). Accessed: Jan 2018.

[65] SmartThings Support. 2018. SmartThings: Local Processing. https://support.smartthings.com/hc/en-us/articles/209979766-Local-processing. (2018). Accessed: March 2018.

[66] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *International Conference on World Wide Web (WWW) 2017*.

[67] Eric A. Taub. 2018. New York Times: Sleepy Behind the Wheel? Some Cars Can Tell. https://www.nytimes.com/2017/03/16/automobiles/wheels/drowsy-driving-technology.html. (2018). Accessed: May 2018.

[68] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, Xiaofeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. SmartAuth: User-Centered Authorization for the Internet of Things. In *USENIX Security Symposium 2017*.

[69] Eran Tromer and Roei Schuster. DroidDisintegrator: Intra-application Information Flow Control in Android Apps. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS) 2012*.

[70] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. Trigger-action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *ACM CHI Conference on Human Factors in Computing Systems 2016*.

[71] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and Logging in the Internet of Things. In *Network and Distributed System Security Symposium (NDSS) 2018*.

[72] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *USENIX Security Symposium 2012*.

[73] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. The Feasibility of Dynamically Granted Permissions: Aligning Mobile Privacy with User Preferences. In *IEEE Symposium on Security and Privacy 2017*.

[74] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a Trillion (Unfixable) Flaws on a Billion Devices: Rethinking Network Security for the Internet-of-things. In *ACM Workshop on Hot Topics in Networks (HotNets) 2015*.

[75] Guangsen Zhang and Manish Parashar. Dynamic Context-Aware Access Control for Grid Applications. In *IEEE International Workshop on Grid Computing (CCGRID) 2003*.

[76] Hong Zhang, Yeping He, and Zhiguo Shi. Spatial Context in Role-based Access Control. In *Springer International Conference on Information Security and Cryptology (ICISC) 2006*.

A ACCESS CONTROL IN IOT

A.1 Direct resource access

As an illustrative example for a resource layer protocol, we focus on the Open Connectivity Foundation (OCF) standard (see Section 5). It defines the functionality of an IoT resource layer.

Subjects. In the OCF environment, subjects are *clients* and *servers*, which are processes running on top of physical devices. A client can be, for example, a hub or a smartphone terminal that controls sensors and actuators using requests. Sensors and actuators (lightbulbs, smoke detectors, motion sensors, etc.) are typically servers. A server can issue requests and is therefore a subject, too. For example, after detecting movement, a motion sensor can send a "turn on" request to a smart lightbulb.

Objects. In OCF, objects are *resources* exported by servers. Clients issue requests to servers, and servers to other servers, to access resources. Locally, a resource is uniquely identified by its URI (e.g.,

/home/living_room/light). Globally, a resource is addressed using a device identity (see below) and a URI.

If a resource is discoverable, its global address can be found by any client or server via broadcast "discover" messages. A parent-child relationship is defined for some resources, forming a hierarchy reflected in their URIs, e.g., a /home/living_room/light resource can represent two devices: /home/living_room/light/led1, and /home/living_room/light/led2.

There are two types of resources. *Configuration resources* store and expose client or server configuration values, identities, cryptographic credentials, and other metadata. They are created automatically by the app framework and have a *resource owner* identity.

Device resources implement arbitrary server functionality and are registered using the app-framework APIs. A device resource exposes five operations: read, write, update, delete, and observe. Typically, device resources call device-specific APIs when accessed by a client (e.g., /home/living_room/light_off invokes a system call on the device that turns off the light).

Requests to remote servers have an associated *client identity*: the owner identity of the ''/doxm'' configuration resource. The *server identity* (or *device identity* in the OCF standard's terminology), also stored in the ''/doxm'' resource, is used when handling incoming requests. Identities are represented as GUIDs. A designated configuration resource in every client and server, /creds, contains remote identities and their associated TLS credentials.

Access-control policies. The configuration resource /acl of a server contains the access-control list (ACL) for the resources managed by the server process. The ACL is a list of ACEs, with each ACE specifying a subject (an identity), an object (the URI of a resource), and permissions. Permissions are specified using a mask of 5 bits corresponding to create, read, update, delete, notify (CRUDN) operations. Resource discovery, as well as read and observe operations, require read permission; write and update operations require write permission; delete, notify, and create operations require delete, notify, and create permissions, respectively.[6] Any access to a resource by its owner (if it is a configuration resource), or by the *device owner* (another identity stored in the ''/doxm'' resource), is automatically permitted.

Policy creation. The device owner and the resource owner of /acl (who may have the same identity) set the policy. When installed into a home network, devices follow a provisioning procedure to determine the initial owner identities. For example, the owner identity can be the client identity of a terminal or a hub device operated by a service (e.g., SmartThings). This way, the service can configure and control resource-layer policies and offer a user-facing interface that abstracts away some of their low-level details.

Access-control enforcement. When a server or a client invokes one of the server resources, the caller's identity is first verified using /cred information. The server then checks in /acl if the caller is allowed to perform the requested operation. If so, the requested operation handler is invoked by the reference monitor.

Situational constraints. There is no support for situational constraints in the OCF standard.

[6]In IoTivity v1.3, there is no use and no implementation for notify and delete operations.

A.2 Service-service interaction

Subjects and objects. Users often have accounts with multiple Web services. Typically, *subject services* issue HTTP requests to *object services* to access APIs associated with a specific user. For example, Nest Web APIs enable third-party services to access Nest devices; these APIs are currently used by over 50 services [47]. Third-party services control Nest devices by issuing requests to URLs of the form https://developer-api.nest.com/devices/DEVICE_TYPE/DEVICE_ID/API_ID.

Access-control policies. Object services typically protect their APIs via OAuth 2 [39] *bearer tokens*. This is a capability-based permissioning system, where capabilities (tokens) give the bearer certain access rights. Bearer tokens have associated user accounts and *scopes*. A scope specifies the token's access rights, encoded as a list of *permission strings*. For example, in Nest, the "camera read" permission string gives access rights to APIs such as is_online, but accessing images requires the "camera read + images" permission.

Policy creation. In OAuth 2, policy creation is an access delegation process where users allow access "on their behalf." This involves the object service securely sharing an access token with a subject service, using a Web flow that includes both services, as well as the user who approves the delegation of access rights [12, 39]. See also Appendix D.

Access-control enforcement. To invoke an API, the subject service passes a token as part of the HTTP request. The request is allowed if the token maps to a user identity, has not expired, and its scope contains an access right string for the requested API call.

Situational constraints. Services can implement additional protections beyond bearer token authorizations, including situational constraints. The Nest framework tracks the "user is home" situation using the phone GPS sensor available to the Nest mobile app. The Nest camera can be configured to turn on or off automatically depending on the detected situation. Moreover, services can turn the camera on only with explicit user consent (involving a prompt from the Nest mobile app). This means that if the camera automatically turns off when the user enters, access to the camera is indiscriminately blocked when the user is at home—unless the user turns the camera on, in which case it is indiscriminately allowed.

Another example of a situational constraint is that even when the camera is on, the user has to opt-in via Nest's configuration to enable live-feed access by any third-party services (short GIF animations and images, however, are not protected by this constraint). In effect, this is a situation ("user allows third-party access") that is explicitly activated and de-activated by the user.

Issues in service-service access control. As demonstrated in Section 2, tracking of situational constraints by Nest and other services is often inadequate. In general, access control in Web services is plagued by many other problems. We do not address them in this paper but mention them here for completeness.

First, with the bulk of IoT-device and Web-account functionality packaged into multiple different services, the user does not have a central interface for viewing, granting, and revoking inter-service permissions. Moreover, granting permissions is easier than revoking them because users are prompted to grant a permission

when configuring the service that requires the permission but never prompted to revoke this permission afterwards.

Second, permissive interfaces and overprivileging are ubiquitous. For example, IFTTT requires services to issue OAuth 2 bearer tokens (or refresh tokens) that never expire, presumably because expiring tokens can make recipes fail. Services often allow *any* other (known, authenticated) service to request access tokens, not just IFTTT. Moreover, it is natural to implement OAuth 2 delegation in a subject-agnostic manner, and therefore IFTTT-compliant services may issue non-expiring tokens indiscriminately, not just to IFTTT. This problem is outside the scope of this paper and we did not measure its prevalence.

Third, user-facing permission descriptions are not standardized. Different services use very different description styles, permission semantics, and permission granularities. If it is hard for users to comprehend Android user prompts [25], it is virtually impossible for them to reason about permissions for Web services, even though they are equally dangerous. For example, users of IFTTT—which interacts with many other services—frequently encounter many different OAuth 2 prompts with different, hard-to-follow semantics. To reduce the number of prompts, IFTTT eagerly requests access to all APIs of a Web-service channel once the user activates a single recipe that uses any of them, resulting in overprivileging,[7] as noted by Fernandes et al. [31]

A.3 App-service interaction

We use Samsung's SmartThings API as an example of app-service interaction.

Subjects. SmartThings apps, *SmartApps*, are essentially collections of event handlers written in the Groovy language. Apps are supplied to users through a curated app market. They typically run on the Samsung cloud service, but some can run offline on the SmartThings hub, and this facility is expected to be enabled for more apps [65].

Objects. SmartApps control smart devices compatible with the SmartThings service and, typically, connected to the SmartThings hub device. The access-control system protects accesses of apps (subjects) to device APIs (objects).

Access-control policies. Apps control devices using their *skills*:[8] collections of commands and attribute values that are exposed by devices. Many skills are defined in the API and they tend to be very specific vs., for example, Android permissions. For example, "dishwasher mode" allows to get and set operation modes for the dishwasher, i.e., Home/Away/Night (see below). Some of the skills are similar to OCF resource types (see Section A.1).

Policy creation. App declare the skills they require, while SmartThings devices declare the skills they implement. When installing a SmartApp, the user is prompted to assign to it devices implementing its required skill(s). The skills requested by a SmartApp do not define its privilege level (and calling them "permissions" may be confusing). Instead, access rights are determined via device assignment by the user: a SmartApp is allowed to access all of its assigned devices' APIs. Fernandes et al. noted that this design often results in overprivileging [28].

Access-control enforcement. The Groovy runtime environment on the SmartThings cloud platform checks resource-access operations against app skills.

Situational constraints. SmartThings tracks three specific situations through *modes*: Home, Away, and Night. Modes are described by the documentation as "behavioral filters." The user can configure apps to run only in specific modes.Apps can (and are encouraged by the developer documentation) change their behavior according to modes. Mode changes are highly configurable using *routines*, user-defined automation rules with triggers and actions (similar in structure to IFTTT rules), and apps. A natural configuration is to switch the Home/Away mode according to the device's location, and set the Night mode according to the time of day. SmartThings users can add custom modes but developers cannot. Modes are mutually exclusive and thus not suitable for tracking multiple non-exclusive situations.

A.4 Channel-recipe interactions (IFTTT)

The IFTTT recipe structure is described in Section 4. In IFTTT, recipes are *subjects*, actions are *objects*. *Access-control policies* are implicitly defined by recipe functionality, i.e., every recipe can access its attached actions (subject to situational constraints, see below). Correspondingly, *policy creation* is the process of activating recipes.

Situational constraints. Recipes can potentially do dangerous things, such as disarm an alarm system or tweet on the user's behalf, but they can fire only if they are activated by their triggers. One can view triggers as having a dual purpose: they define the sufficient and necessary conditions for the execution of recipe actions, or, in other words, the situation in which the trigger executes.

In IFTTT, every recipe is limited to one situational constraint defined by its trigger, as well as two other optional constraints. First, for any activated recipe, users can choose to be notified when it runs. Notifications are transmitted through push messages to the IFTTT app on the user's phone. The second constraint is recipe *filters*, which are pieces of code that execute after the trigger and prior to the action. A filter can "decide," based on the situational factors such as the time of day or information available about the trigger-firing event, not to execute the action. For example, a developer can define that a recipe only runs between 2pm and 4pm.

A planned future feature is the ability to *query* third-party interfaces from the filter code. With this ability, filters would be a powerful tool for enforcing situational constraints. In contrast to our ESOs, however, filters cannot be shared among recipes.

B IFTTT SURVEY

Collected data. We implemented a crawler that extracts IFTTT recipes, actions, and triggers, similarly to Ur et al. [70] and Surbatovich et al. [66]. We extracted the recipes, actions, and triggers of the 571 recipe collections curated by IFTTT *partner services*, accessible from the "services" page, https://ifttt.com/search/services, as

[7]For example, if the user activates a recipe that requires access to any Facebook channel, even a read-only one, IFTTT will immediately ask for access to the following Facebook Web API access rights, which include the right to post to the user's page [20]: `manage_pages, public_profile, publish_actions, user_about_me, user_events, user_friends, user_location, user_photos, user_posts, user_status, user_website`.

[8]"Capabilities" in SmartThings terminology, but we use "skills" to avoid confusion with the standard access-control meaning of "capability."

of January 5th, 2018. We extracted 13619 recipes,[9] not counting the "notify me about new recipes for this service" recipe. It exists for every service and we removed it from our analysis.

Our statements about IFTTT recipes in Section 2 are based on the following recipe statistics:

Top 10 most utilized triggers (by recipes).

(1) say a specific phrase by AMAZON_ALEXA (498 recipes)
(2) button press by DO_BUTTON (457 recipes)
(3) you enter an area by LOCATION (309 recipes)
(4) you exit an area by LOCATION (239 recipes)
(5) every day at by DATE_AND_TIME (234 recipes)
(6) new public video uploaded by you by YOUTUBE (215 recipes)
(7) say a simple phrase by GOOGLE_ASSISTANT (200 recipes)
(8) every day of the week at by DATE_AND_TIME (155 recipes)
(9) any new note by DO_NOTE (131 recipes)
(10) flic is clicked by FLIC (112 recipes)

Top 10 most utilized actions (by recipes).

(1) send a notification from the ifttt app by IF_NOTIFICATIONS (823 recipes)
(2) add row to spreadsheet by GOOGLE_SHEETS (648 recipes)
(3) send me an email by EMAIL (491 recipes)
(4) send an email by GMAIL (332 recipes)
(5) post a tweet by TWITTER (322 recipes)
(6) post to channel by SLACK (277 recipes)
(7) quick add event by GOOGLE_CALENDAR (245 recipes)
(8) change color by HUE (192 recipes)
(9) call my phone by PHONE_CALL (187 recipes)
(10) turn on lights by HUE (181 recipes)

Top 10 most utilized triggers (by users).

(1) button press by DO_BUTTON (2904292 users)
(2) any new photo by you by INSTAGRAM (2296873 users)
(3) say a specific phrase by AMAZON_ALEXA (2088880 users)
(4) you enter an area by LOCATION (1933872 users)
(5) new feed item by FEED (1708995 users)
(6) any new note by DO_NOTE (1401746 users)
(7) every day at by DATE_AND_TIME (1261117 users)
(8) any new photo by DO_CAMERA (1196368 users)
(9) tomorrow's forecast calls for by WEATHER (1194811 users)
(10) you exit an area by LOCATION (1022967 users)

Top 10 most utilized actions (by users).

(1) send a notification from the ifttt app by IF_NOTIFICATIONS (6830526 users)
(2) send me an email by EMAIL (5282087 users)
(3) add row to spreadsheet by GOOGLE_SHEETS (4760448 users)
(4) quick add event by GOOGLE_CALENDAR (2460103 users)
(5) post a tweet by TWITTER (1868969 users)
(6) upload file from url by GOOGLE_DRIVE (1446966 users)
(7) add file from url by DROPBOX (1365485 users)
(8) post a tweet with image by TWITTER (1318712 users)
(9) create a note by EVERNOTE (1225323 users)
(10) call my phone by PHONE_CALL (1178089 users)

Home/away trigger. We observe that location-based triggers are among the most common, with over 500 apps and 3 million users using them. The "location" trigger is provided by the IFTTT service itself, not by a partner service. It uses the location capabilities of the user's phone to determine the area the user is in. To use it, the IFTTT mobile app must be installed.

We manually examined the descriptions of 100 randomly chosen apps that use location-based triggers (enter an area, leave an area, enter or leave an area) to understand why they use location. Most recipes use location for determining if the user is in a specific *place*: home for 75, gym for 5, work for 2, and vacation for 2. Only 16 recipes do not specify the meaning of the location-based trigger. For the majority of those, however, it appears that location is most likely used for determining if the user is at home (e.g., "disarm your alarm when you leave an area").

C ESO EXAMPLES

To illustrate the range of situations that are useful for IoT access control, we suggest (but do not implement) several ESOs in addition to those described in Section 6.

User is away, using GPs and presence sensors. `{situation_name: HomeAway; ESO_description: "Home presence detection using phone geolocation, Nest and SmartThings presence sensors."}` **Protected APIs, configuration:** user is required to install an Android app that uses location services, configure their location, and indicate which sensors to use. If Nest is used, the ESO asks for the Nest API token `away` and calls into the Nest API for the status of the presence sensor, https://developer-api.nest.com/structures/home_id/away[10] (see Section A.2). Optionally, the user installs a SmartThings SmartApp which requests the `presenceSensor` capability (see Section A.3), and (upon installation) asks the user for access to all presence sensors in the home. **Behavior:** in addition to geofencing-based tracking, this ESO also uses the presence-sensor data from Nest and SmartThings. If presence was recently detected and geofencing does not indicate that the user is at home, then the phone app starts to actively query the location API (as opposed to passively waiting for an "enter area" event). *is_active*() returns "active" if the user is *not* at home, "not active" otherwise.

Notify user via push notifications. `{situation_name: NotifyUser; ESO_description: "whenever access occurs, send the user a push notification with the access details."}` **Protected APIs„ configuration:** user is required to install an Android app but does not need to approve any permissions beyond normal ones needed for all apps. **Behavior:** *is_active*(s, a) sends the user a push notification if there has been an access of subject s with capability a. Always returns "active".

Notify user via push notifications and smart lights. `{situation_name: NotifyUser; ESO_description: "Whenever`

[9] The URL access used by [70] and [66] to exhaustively crawl all recipes, including those created by users and not advertised in channel pages, is no longer available. However, the vast majority of the recipes collected in these studies are duplicates and unpopular recipes. Our collection contains about 6,000 unique recipes, compared to Ur et al.'s 16,000. The mean user adoption of recipes in our collection is 4,884 users and the median is about 55 users, vs. Ur et al.'s mean of 52 and median of 1. This is at least partly due to the growing adoption of IFTTT in general after the Ur et al. collection was published.

[10] This Nest API itself uses phone geolocation in addition to Nest sensors. Nest currently does not offer third-party access to "raw" presence sensors. However, when the phone location sensors indicate the user is away, and the in-house presence sensors detect occupancy, this API will return an indication, which is useful for this ESO.

access occurs, send the user a push notification and blink smart lights."} **Protected APIs, configuration:** user is required to install an Android app (none of its permissions require user authorization) and a SmartThings app that uses the `light` capability. **Behavior:** *is_active*(s, a) sends the user a push notification and logs the fact that there has been an access of subject s with capability a. Always returns "active".

Use cases of "notify user" ESOs. This situation is useful in many diverse, common scenarios. Our study of IFTTT recipes (see Section 2.2) indicates that many users would like to be notified of almost any possibly dangerous operation. Automatic user notification is natural for accesses that are (a) infrequent and (b) require the user's attention anyway, e.g., taking pictures if a burglary has been detected or a smart garage door opened when the user is away.

D OAUTH 2

OAuth 2access delegation flows. Delegation involves the user, a user agent (e.g., browser, app), and the subject and object services. The delegation flow begins when (1) the user-agent issues a request to the subject service. Then, (2) user-agent is redirected by the subject service to the object service. A request for a token with a specified scope can be transmitted over the redirect request. Then, (3) the object service "obtains an authorization decision (by asking the resource owner or by establishing approval via other means)" [39], and (4) redirects the user agent back to the subject service. Then, (5) the access token is passed to the subject service either over the final redirect request[11] (*implicit grant*) or using direct service-service communication (*authorization code token grant*). The object service can return an access token with a scope different from the requested scope (specifying the de-facto granted scope in the "access token response" that relays the token from the object to the subject). Finally, (6) the subject can access the protected resource. Figure D.1 depicts the access delegation flow.

Optionally, a *refresh token* can be granted along with an access token. The refresh token has a later expiry date and can be used to fetch a fresh access token to replace an expired one.

Example: adding a Nest Cam Alexa skill To illustrate the relevant OAuth 2functionality, consider a user who wishes to enable the "Nest Cam" Alexa skill. This skill lets users control Nest Cam via Amazon products, e.g., stream the camera feed directly to Amazon Show, which is an Alexa terminal with a screen. The user can install this skill from a skills market on the Amazon Web site. Since Alexa accesses Nest Cam through its Web API, it requires the corresponding OAuth 2token. If it does not have the token already, it initiates an OAuth 2flow to obtain the token. The Nest service then authenticates the user (e.g., using a password or an authentication cookie) and prompts her for approval. When the flow ends and if permission was granted, the user is redirected back to the skills market where she can continue to configure the skill.

On-demand permissioning. OAuth 2is popular because it is easy for users and developers. OAuth 2delegation flows can be seamlessly nested in any flow of the subject service where the user is involved. They can begin whenever the user invokes an operation in the subject service that requires access to a third-party service. When a delegation flow ends, the user is redirected back to continue the original operation. In our example, when the flow ends, if the user has approved Amazon's access to the camera in *some* Nest account and was redirected back to Amazon, the configuration flow can continue seamlessly.

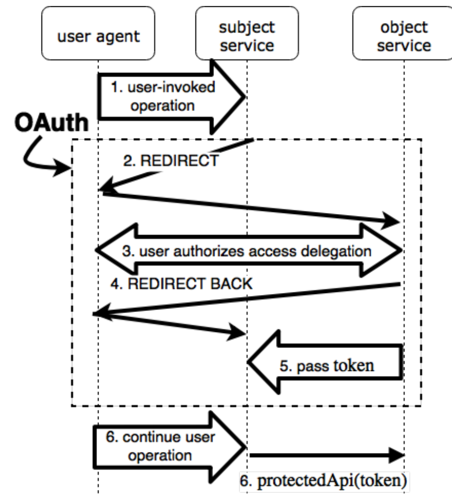


Figure D.1: OAuth 2access delegation

User approval and permission semantics. The process of obtaining the user's approval is not standardized in OAuth 2. Typically, following user identification and authentication, the object website displays an HTTP form to request the permission grant. It includes a description of the requested permissions and the URL identifying Amazon, and prompts the user for approval.

[11] Not directly over the request but via an HTTP `fragment` field, obtainable by a service script—see OAuth 2specification [39].